MiniSoft Compiler Implementation

HADJ ARAB Adel RACHEDI Abderrahmane

April 20, 2025

Abstract

This report details the design and implementation of a compiler for the MiniSoft language using Rust with LALRPOP for syntax analysis and Logos for lexical analysis. The compiler supports the full MiniSoft language specification and implements all compilation phases from lexical analysis to code generation.

Contents

1 Introduction

1.1 Project Overview

This project involves the development of a compiler for the MiniSoft programming language, implemented using the Rust programming language. A compiler is a specialized software tool that translates human-readable source code into machine-executable instructions. The MiniSoft compiler represents an implementation that handles four phases of the compilation process, from initial code analysis to the generation quadruples.

The primary goal of this project is to create a fully functional compiler that correctly implements the MiniSoft language specification while demonstrating modern compiler design principles. This includes robust error handling, and a modular architecture that separates the compilation process into distinct phases.

1.2 MiniSoft Language Features

MiniSoft is a compact programming language designed for educational purposes, combining simplicity with a range of essential programming features:

- **Variables and Data Types**: MiniSoft supports three primary data types: integers (from -32768 to 32767), floating-point numbers (decimals).
- **Arrays**: The language supports single-dimensional arrays for each basic data type.
- **Expressions and Operators**: MiniSoft includes operators for arithmetic calculations, comparisons, and logical operations.
- **Control Flow Structures**: Programmers can use conditional statements (if-then-else) and iterative constructs (do-while and for loops).
- **Constants**: The language allows the definition of named constants.
- **Input/Output Operations**: MiniSoft provides basic facilities for console I/O.
- **Type System**: MiniSoft employs a static type system, with variable types determined at compile time.

Each MiniSoft program consists of a main program block with variable declarations followed by executable statements.

1.3 Tools and Technologies

The MiniSoft compiler is built using modern tools that enable efficient implementation:

- **Rust Programming Language**: Chosen for its memory safety, performance, and pattern matching capabilities.
- **Logos**: A high-performance lexer generator for Rust with attribute-based syntax for defining tokens.
- **LALRPOP**: A parser generator for Rust that allows expressing grammar rules in a declarative forma parser generator for Rust that uses the LR(1) parsing algorithm, enabling developers to define grammar rules declaratively.
- **Cranelift**: A code generator framework used to produce optimized machine code.

Figure 1: MiniSoft Compiler Architecture showing the flow from source code through lexical analysis, syntax analysis, semantic analysis, and code generation.

2 Compiler Design

2.1 Compilation Pipeline

The MiniSoft compiler follows the classical compiler pipeline architecture, divided into sequential phases:

1. **Lexical Analysis (Scanning)**: Reads source code character by character and groups characters into tokens such as keywords, identifiers, literals, and operators.
2. **Syntax Analysis (Parsing)**: Analyzes the sequence of tokens to determine if they follow the MiniSoft grammar rules, building an Abstract Syntax Tree (AST).
3. **Semantic Analysis**: Checks whether the program makes logical sense, including type checking, scope validation, and constant analysisvalidates program logic through declaration, statement, and expression analysis, covering type checking, scope rules, and constant evaluation..
4. **Intermediate Code Generation**: Translates the AST into an intermediate representation using quadruples.

2.2 Architecture Overview

The MiniSoft compiler uses a modular architecture with components that correspond to the phases of compilation:

- **Core Compiler Driver**: Orchestrates the compilation process and handles high-level error reporting.
- **Lexer Module**: Implements lexical analysis using Logos to convert source text into tokens.
- **Parser Module**: Uses LALRPOP to implement syntax analysis, converting tokens into AST.
- **Semantic Analyzer**: Performs type checking, scope analysis, and other semantic validations.
- **Code Generator**: Translates validated AST into executable code.
- **Error Handling System**: Provides unified error reporting across all compiler phases.

2.3 Design Decisions

Key design decisions that shaped the implementation include:

- **Strong Error Reporting**: Prioritizing comprehensive error detection and clear, actionable messages.
- **Location-Aware AST**: Each node carries source location information for precise error reporting.
- **Type Safety Through Rust**: Leveraging Rust's type system to prevent implementation errors.
- **Progressive Validation**: Each compilation phase assumes previous validations have passed.
- **Early Error Detection**: Detecting potential runtime errors at compile time when possible.
- **Intermediate Representation Choice**: Using quadruples for their simplicity and expressiveness.
- **Declaration-Before-Use Requirement**: Requiring variable declarations before use to simplify analysis.

These decisions reflect a balance between educational value, implementation practicality, and user experience.

3 Lexical Analysis

3.1 Overview of Lexical Analysis

Lexical analysis, the first phase of compilation, transforms source code into tokens—the smallest meaningful units of a programming language. In the MiniSoft compiler, this phase is implemented using Logos, which combines declarative syntax with efficient processing.

The lexical analyzer (lexer) scans the input character by character, following two core rules:

- **Longest Match Rule**: Prioritizes the longest possible valid token (e.g., `>=` over `>`).
- **Priority Rule**: Resolves ambiguities by matching the first declared pattern (e.g., keywords before identifiers).

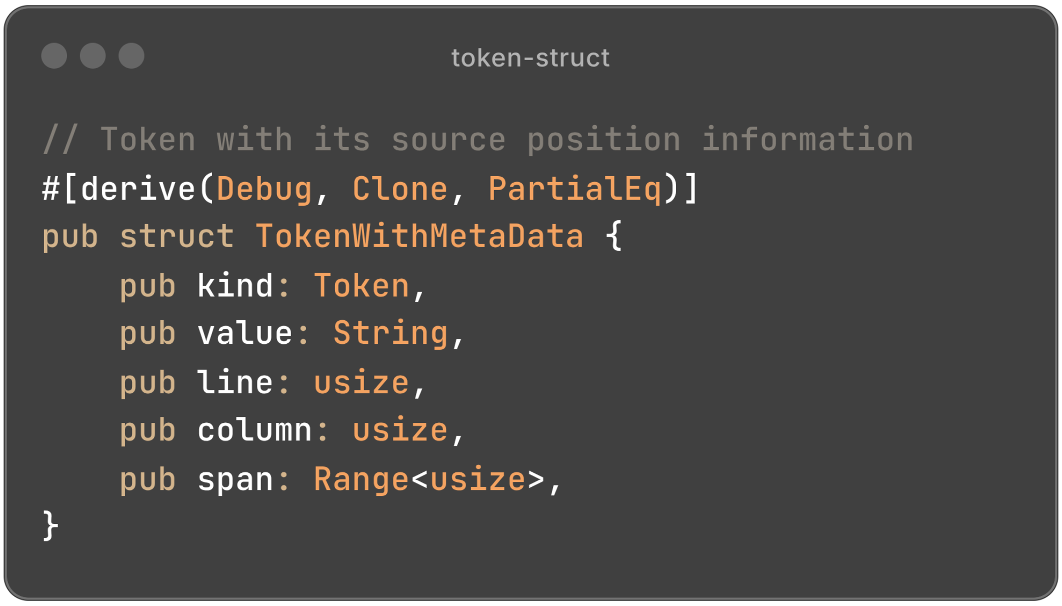
The lexer recognizes token patterns while filtering out non-essential elements like whitespace and comments.

3.2 Token Design

The MiniSoft language employs a comprehensive token classification system:

- **Keywords**: Reserved words `MainPrgm`, `Var`, `BeginPg`, `EndPg`, `let`, `Int`, `Float`, `if`, `then`, `else`, `while`, `for`, `do`, `from`, `to`, `step`, `input`, `output`, `@define`, `Const`
- **Control flow structures**: Tokens `if`, `then`, `else`, `while`, `for`, `do`, `from`, `to`, `step`
- **Declarations**: Tokens `let`, `Int`, `Float`, `Const`
- **Program structure**: Tokens `MainPrgm`, `Var`, `BeginPg`, `EndPg`
- **Operators**: Arithmetic `+`, `-`, `*`, `/`, comparison `>`, `<`, `>=`, `<=`, `==`, `!=`, and logical operators `AND`, `OR`.
- **Punctuation**: Symbols `;`, `,`, `:`, `[`, `]`, , , `(`, `)`
- **Literals**: Integer, floating-point, and string values
- **Identifiers**: User-defined names

Each token carries metadata including the original text, line number, column position, and character span.



```rust
// Token with its source position information
#[derive(Debug, Clone, PartialEq)]
pub struct TokenWithMetaData {
    pub kind: Token,
    pub value: String,
    pub line: usize,
    pub column: usize,
    pub span: Range<usize>,
}
```

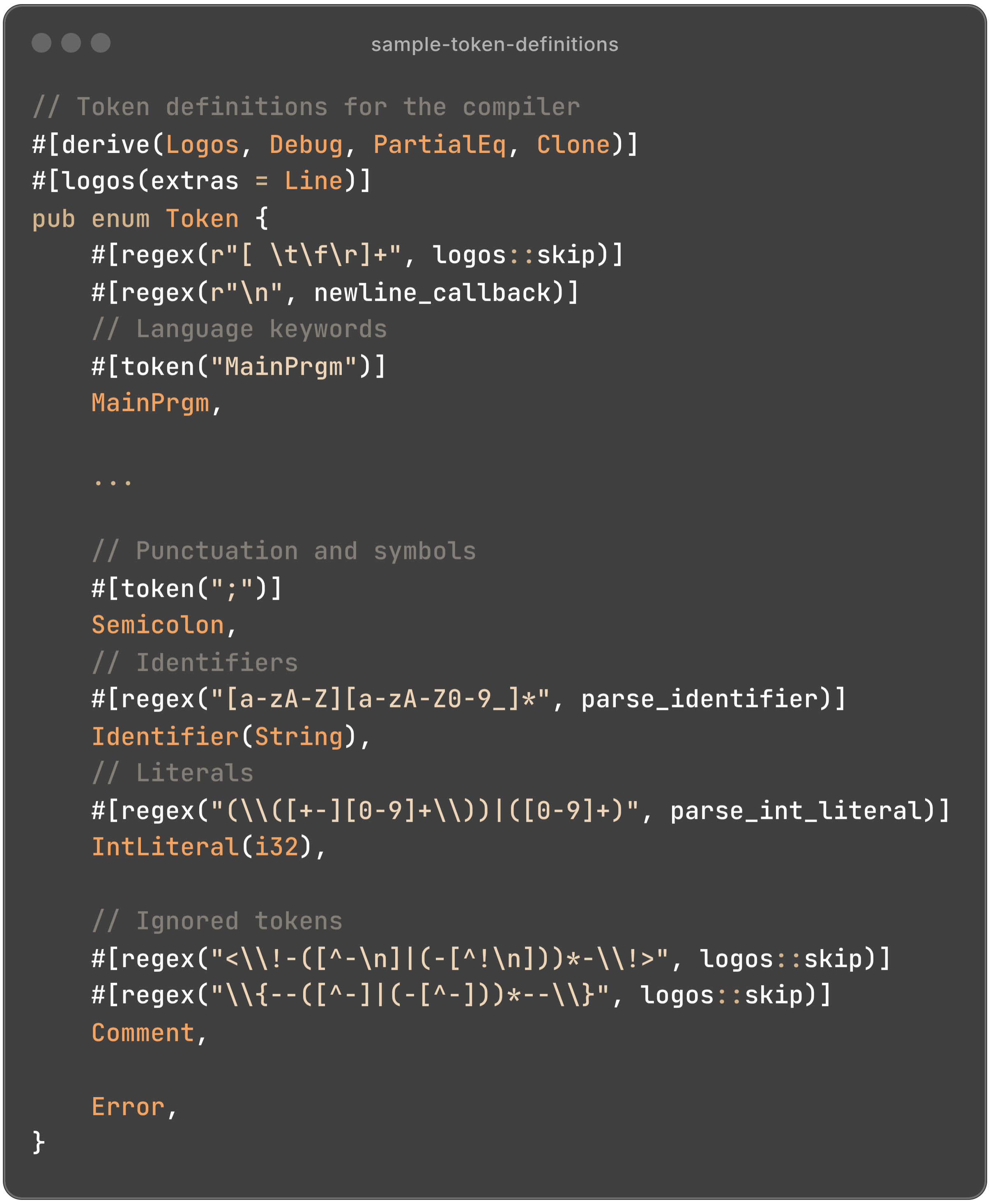
Figure 2: Token Structure

3.3 Lexer Implementation with Logos

The MiniSoft lexer uses Logos' declarative approach to define token patterns:

Listing 1: Logos Token Definitions Syntax

```
#[pattern_type("matching_rule", processing_callback)]
TokenVariant(OutputType)
```



```
// Token definitions for the compiler
#[derive(Logos, Debug, PartialEq, Clone)]
#[logos(extras = Line)]
pub enum Token {
    #[regex(r"[ \t\f\r]+", logos::skip)]
    #[regex(r"\n", newline_callback)]
    // Language keywords
    #[token("MainPrgm")]
    MainPrgm,

    ...

    // Punctuation and symbols
    #[token(";")]
    Semicolon,
    // Identifiers
    #[regex("[a-zA-Z][a-zA-Z0-9_]*", parse_identifier)]
    Identifier(String),
    // Literals
    #[regex("(\\([+-][0-9]+\\))|([0-9]+)", parse_int_literal)]
    IntLiteral(i32),

    // Ignored tokens
    #[regex("<\\!-([^-\n]|(-[^!\n]))*-\\!>", logos::skip)]
    #[regex("\\{--([^-]|(-[^-]))*--\\}", logos::skip)]
    Comment,

    Error,
}
```

Figure 3: Sample Token Definitions

This approach makes the lexer's behavior clear and maintainable, with different token types handled through specific patterns and callbacks.

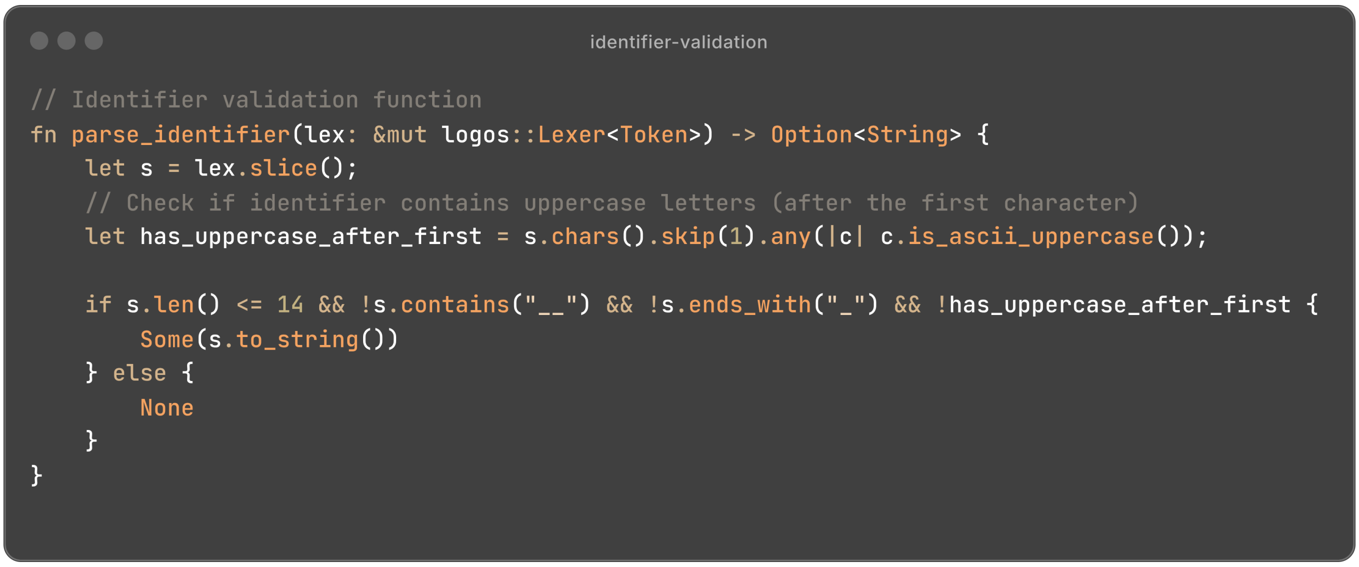
3.4 Handling Special Cases

The MiniSoft lexer implements careful processing for language-specific requirements:

3.4.1 Identifiers with Rules

Identifiers in MiniSoft have specific constraints:

- Maximum length of 14 characters
- No consecutive underscores
- No trailing underscores
- Only the first character may be uppercase
- Must start with a letter

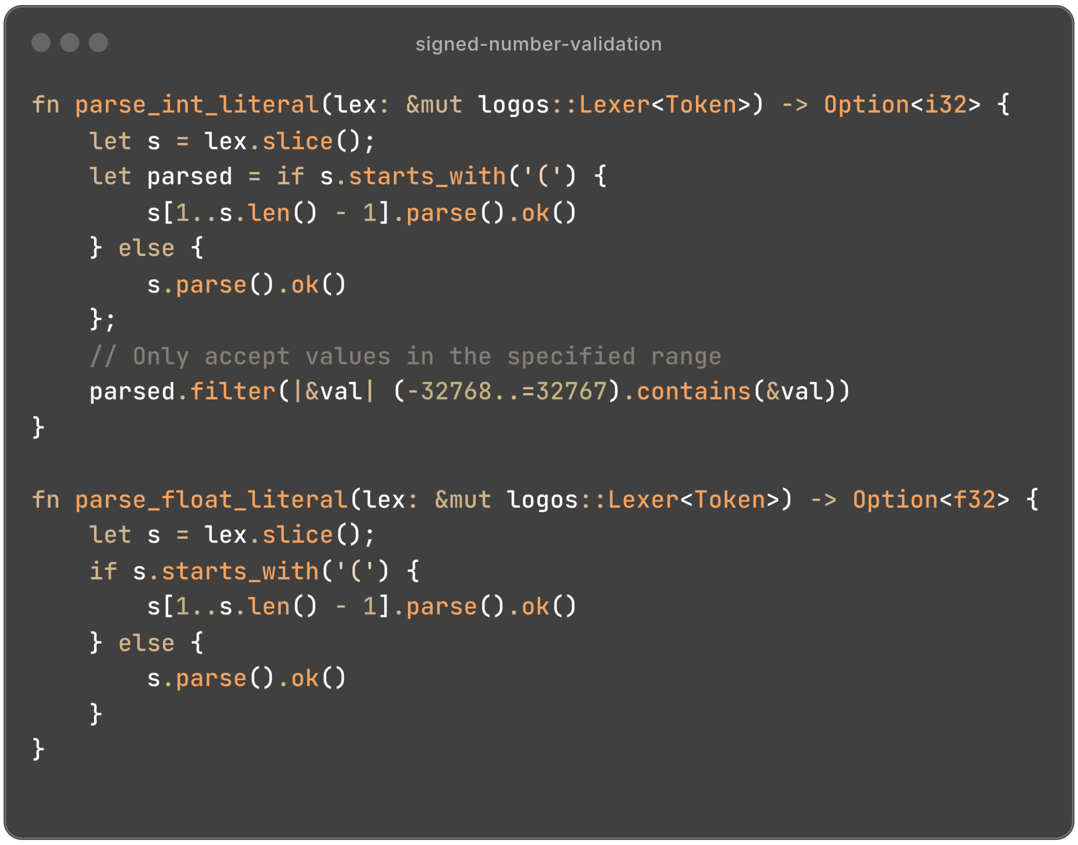


```rust
// Identifier validation function
fn parse_identifier(lex: &mut logos::Lexer<Token>) -> Option<String> {
    let s = lex.slice();
    // Check if identifier contains uppercase letters (after the first character)
    let has_uppercase_after_first = s.chars().skip(1).any(|c| c.is_ascii_uppercase());

    if s.len() <= 14 && !s.contains("__") && !s.ends_with("_") && !has_uppercase_after_first {
        Some(s.to_string())
    } else {
        None
    }
}
```

Figure 4: Identifier Validation

3.4.2 Numbers with Sign

MiniSoft supports parsing of signed integer and floating-point literals. The following code demonstrates the implementation:



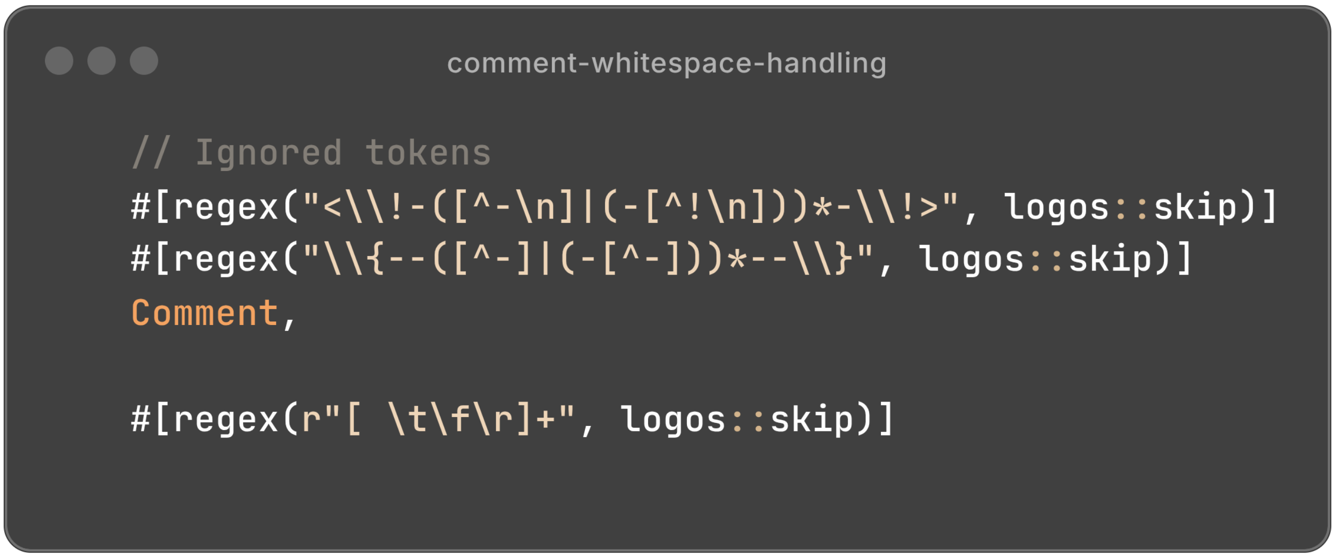
```rust
fn parse_int_literal(lex: &mut logos::Lexer<Token>) -> Option<i32> {
    let s = lex.slice();
    let parsed = if s.starts_with('(') {
        s[1..s.len() - 1].parse().ok()
    } else {
        s.parse().ok()
    };
    // Only accept values in the specified range
    parsed.filter(|&val| (-32768..=32767).contains(&val))
}

fn parse_float_literal(lex: &mut logos::Lexer<Token>) -> Option<f32> {
    let s = lex.slice();
    if s.starts_with('(') {
        s[1..s.len() - 1].parse().ok()
    } else {
        s.parse().ok()
    }
}
```

Figure 5: Identifier Validation

3.4.3 Comments and Whitespace

MiniSoft supports two comment styles:

- C-style comments: `{-comment-}`
- XML-style comments: `<!- comment ->`

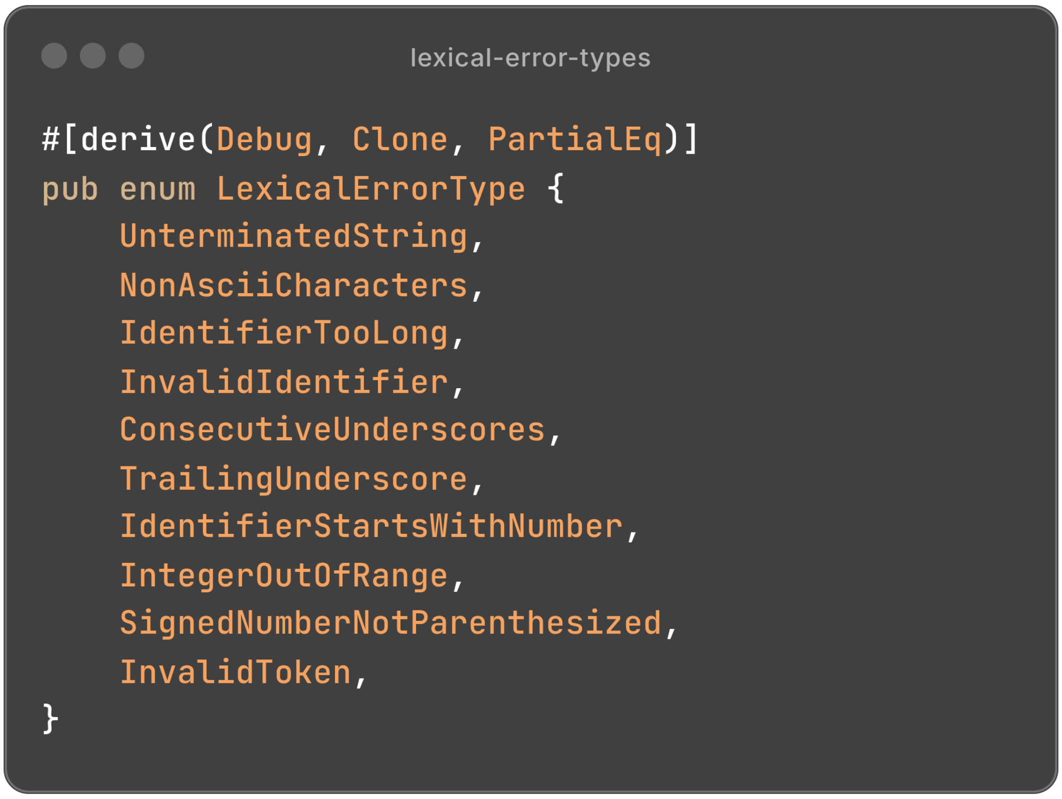


```rust
// Ignored tokens
#[regex("<\\!-([^-\n]|(-[^!\n]))*-\\!>", logos::skip)]
#[regex("\\{--([^-]|(-[^-]))*--\\}", logos::skip)]
Comment,

#[regex(r"[ \t\f\r]+", logos::skip)]
```

Figure 6: Comment and Whitespace Handling

3.5 Error Handling and Reporting

The lexer detects and aggregates all lexical errors during tokenization, halting compilation upon encountering any such errors to ensure early failure.


lexical-error-types

```rust
#[derive(Debug, Clone, PartialEq)]
pub enum LexicalErrorType {
    UnterminatedString,
    NonAsciiCharacters,
    IdentifierTooLong,
    InvalidIdentifier,
    ConsecutiveUnderscores,
    TrailingUnderscore,
    IdentifierStartsWithNumber,
    IntegerOutOfRange,
    SignedNumberNotParenthesized,
    InvalidToken,
}
```

Figure 7: Lexical Error Types

When errors are found, the lexer creates detailed messages with exact positions, problematic text, and suggestions for fixing issues (an example of error output can be found in the error handling section).

4 Syntax Analysis

4.1 Overview of Syntax Analysis

Syntax analysis determines whether tokens form valid language constructs according to the grammar rules. In the MiniSoft compiler, this phase is implemented using LALRPOP, which allows expressing grammar rules in a readable format.

The parser verifies that tokens follow grammatical rules, detects syntax errors, and organizes tokens into an Abstract Syntax Tree (AST).

4.2 Grammar for Syntax Analysis

The MiniSoft grammar is formally defined as LR(1) production rules with embedded semantic actions, ensuring unambiguous parsing and full language coverage.

$$G = (N, T, S, P)$$

Components

- **Non-Terminals (N):** Grammar variables

- **Terminals (T):** Language tokens
- **Start Symbol (S):** Program
- **Productions (P):** Derivation rules

Formal Definition

Non-Terminals:

N = {Program, Declarations, Declaration, TypeSpec, Scope, Statement, LValue,
Expression, LogicalExpr, ComparisonExpr, AdditiveExpr,
MultiplicativeExpr, UnaryExpr, PrimaryExpr, LiteralValue,
IdList, ExprList, OutputExprList, OutputExpression,
OutputAdditiveExpr, OutputMultiplicativeExpr, OutputPrimaryExpr}

Terminals:

T = {“MainPrgm”, “Var”, “BeginPg”, “EndPg”, “let”, “@define”, “Const”, “if”,
“then”, “else”, “do”, “while”, “for”, “from”, “to”, “step”, “input”, “output”,
“Int”, “Float”, *Id*, *Int*, *Float*, *String*,
“;”, “,” , “:”, “[”, “]”, “(”, “)”, “{”, “}”, “:=”, “=”, “+”, “-”,
“*”, “/”, “>”, “<”, “>=”, “<=”, “==”, “!=”, “AND”, “OR”, “!”}

Production Rules:

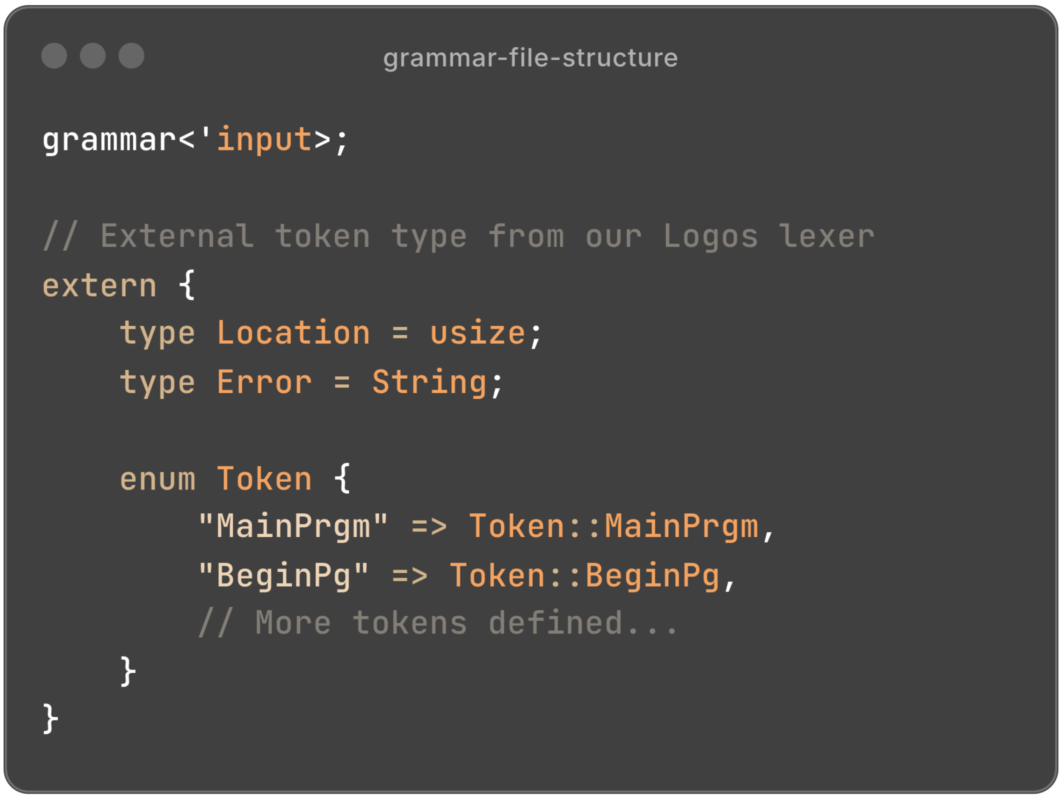
Program	→ “MainPrgm” *Id* “;” “Var” Declarations “BeginPg” Scope “EndPg” “;”
Declarations	→ Declaration*
Declaration	→ “let” IdList “:” TypeSpec “;”
	| “let” IdList “:” “[” TypeSpec “;” *Int* “]” “;”
	| “let” IdList “:” TypeSpec “=” Expression “;”
	| “let” IdList “:” “[” TypeSpec “;” *Int* “]” “=” “{” ExprList “}” “;”
	| “@define” “Const” *Id* “:” TypeSpec “=” LiteralValue “;”
TypeSpec	→ “Int” | “Float”
Scope	→ “{” Statement* “}”
Statement	→ LValue “:=” Expression “;”
	| “if” “(” Expression “)” “then” Scope[“else” Scope]
	| “do” Scope “while” “(” Expression “)” “;”
	| “for” LValue “from” Expression “to” Expression “step” Expression Scope
	| “input” “(” LValue “)” “;”
	| “output” “(” OutputExprList “)” “;”
LValue	→ *Id* | *Id* “[” Expression “]”
Expression	→ LogicalExpr
LogicalExpr	→ ComparisonExpr{(“OR” | “AND”) ComparisonExpr}
ComparisonExpr	→ AdditiveExpr{(“==” | “!=” | “<” | “>” | “<=” | “>=”)AdditiveExpr}
AdditiveExpr	→ MultiplicativeExpr{(“+” | “-”) MultiplicativeExpr}
MultiplicativeExpr	→ UnaryExpr{(“*” | “/”) UnaryExpr}
UnaryExpr	→ [“!”]PrimaryExpr
PrimaryExpr	→ *Id* | *Id* “[” Expression “]” | LiteralValue | “(” Expression “)”
LiteralValue	→ *Int* | *Float* | *String*
IdList	→ *Id*{“,” *Id*}
ExprList	→ Expression{“,” Expression}
OutputExprList	→ [OutputExpression{“,” OutputExpression}]
OutputExpression	→ *String* | OutputAdditiveExpr
OutputAdditiveExpr	→ OutputMultiplicativeExpr{(“+” | “-”) OutputMultiplicativeExpr}
OutputMultiplicativeExpr	→ OutputPrimaryExpr{(“*” | “/”) OutputPrimaryExpr}
OutputPrimaryExpr	→ *Id* | *Id* “[” Expression “]” | LiteralValue | “(” OutputAdditiveExpr “)”

Key Features

- LR(1) parsing compatible
- Type-aware productions
- Explicit scoping rules
- Complete expression hierarchy
- Array support in declarations
- Input/output operations
- Control flow constructs
- Constant definitions

4.3 Grammar Specification with LALRPOP

LALRPOP uses LR(1) parsing techniques to analyze program structure. The grammar for MiniSoft is specified in a declarative format.

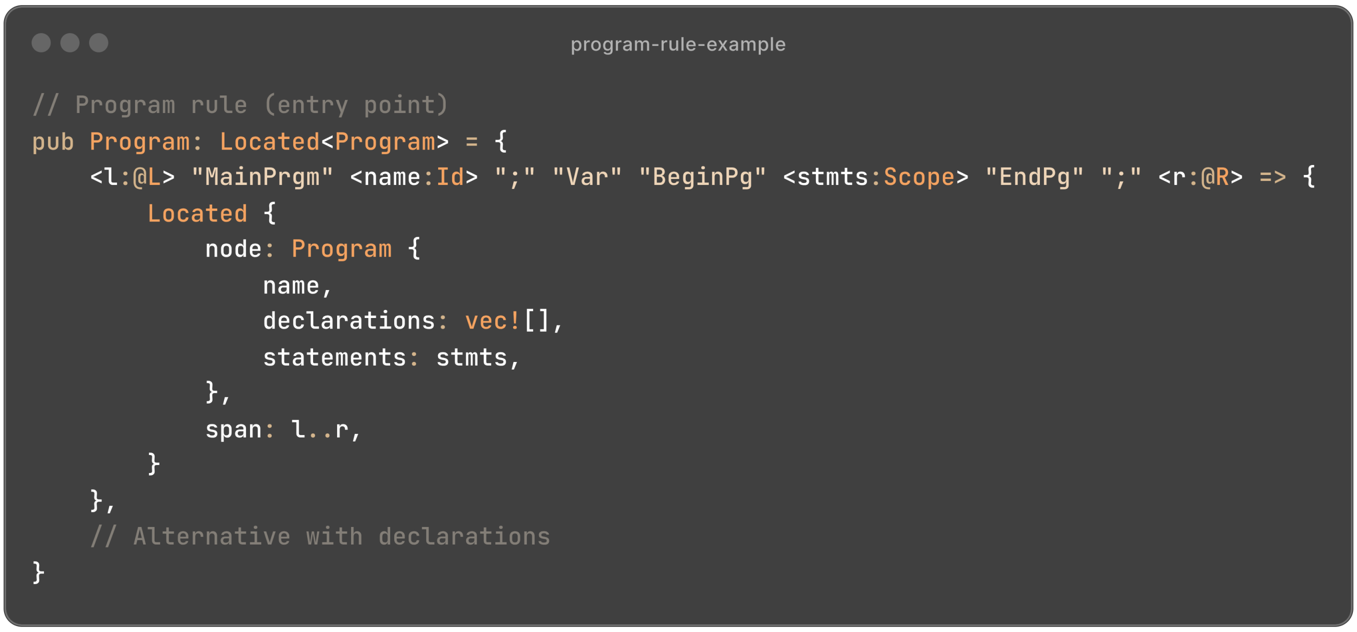


grammar-file-structure

```
grammar<'input>;

// External token type from our Logos lexer
extern {
    type Location = usize;
    type Error = String;

    enum Token {
        "MainPrgm" => Token::MainPrgm,
        "BeginPg" => Token::BeginPg,
        // More tokens defined...
    }
}
```

Figure 8: Grammar File Structure

Production rules define how language constructs are formed from simpler elements:

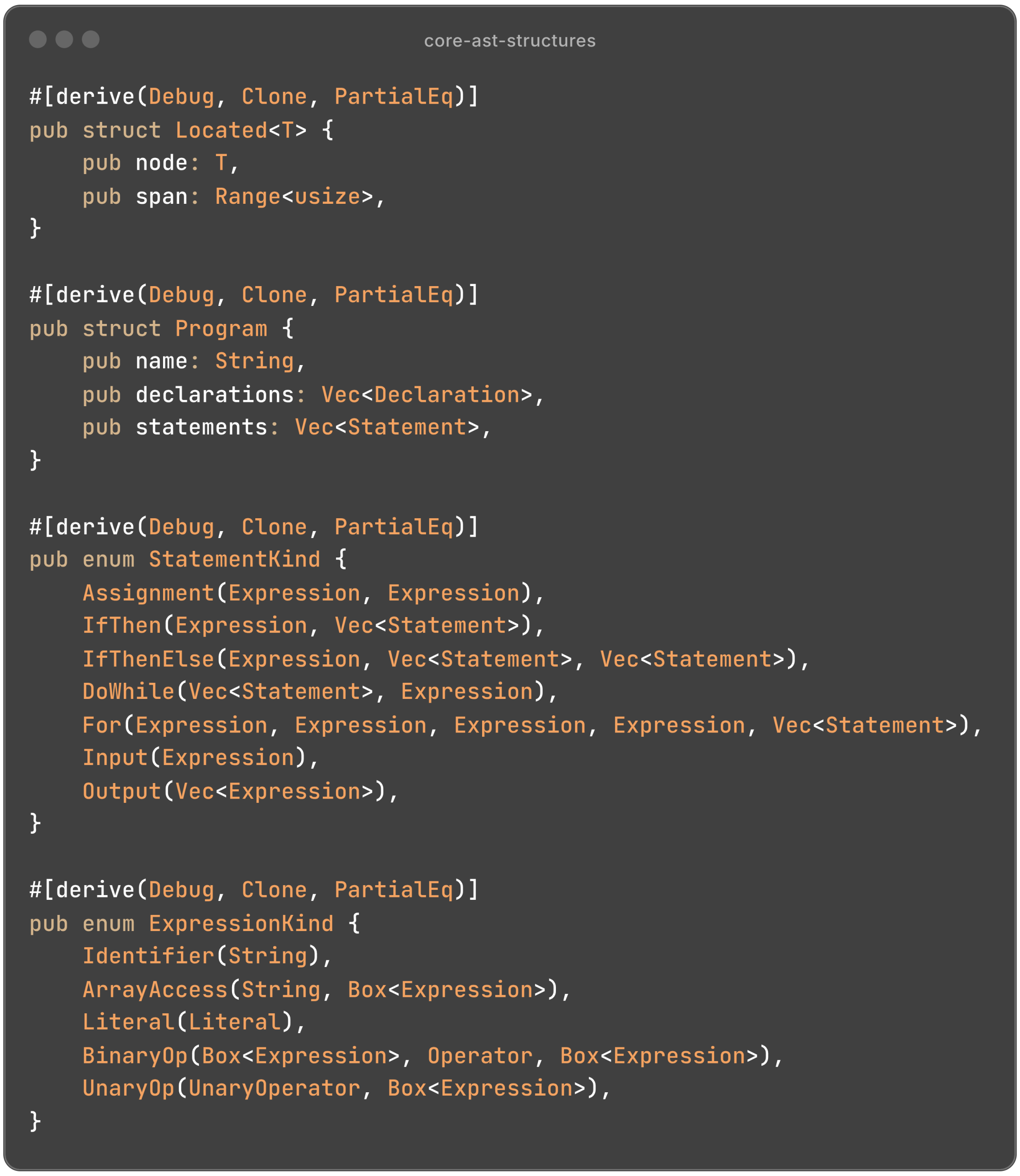


```
// Program rule (entry point)
pub Program: Located<Program> = {
    <l:@L> "MainPrgm" <name:Id> ";" "Var" "BeginPg" <stmts:Scope> "EndPg" ";" <r:@R> => {
        Located {
            node: Program {
                name,
                declarations: vec![],
                statements: stmts,
            },
            span: l..r,
        }
    },
    // Alternative with declarations
}
```

Figure 9: Program Rule Example

4.4 Abstract Syntax Tree Design

The Abstract Syntax Tree (AST) is a hierarchical representation of the program that serves as the foundation for subsequent compilation phases.



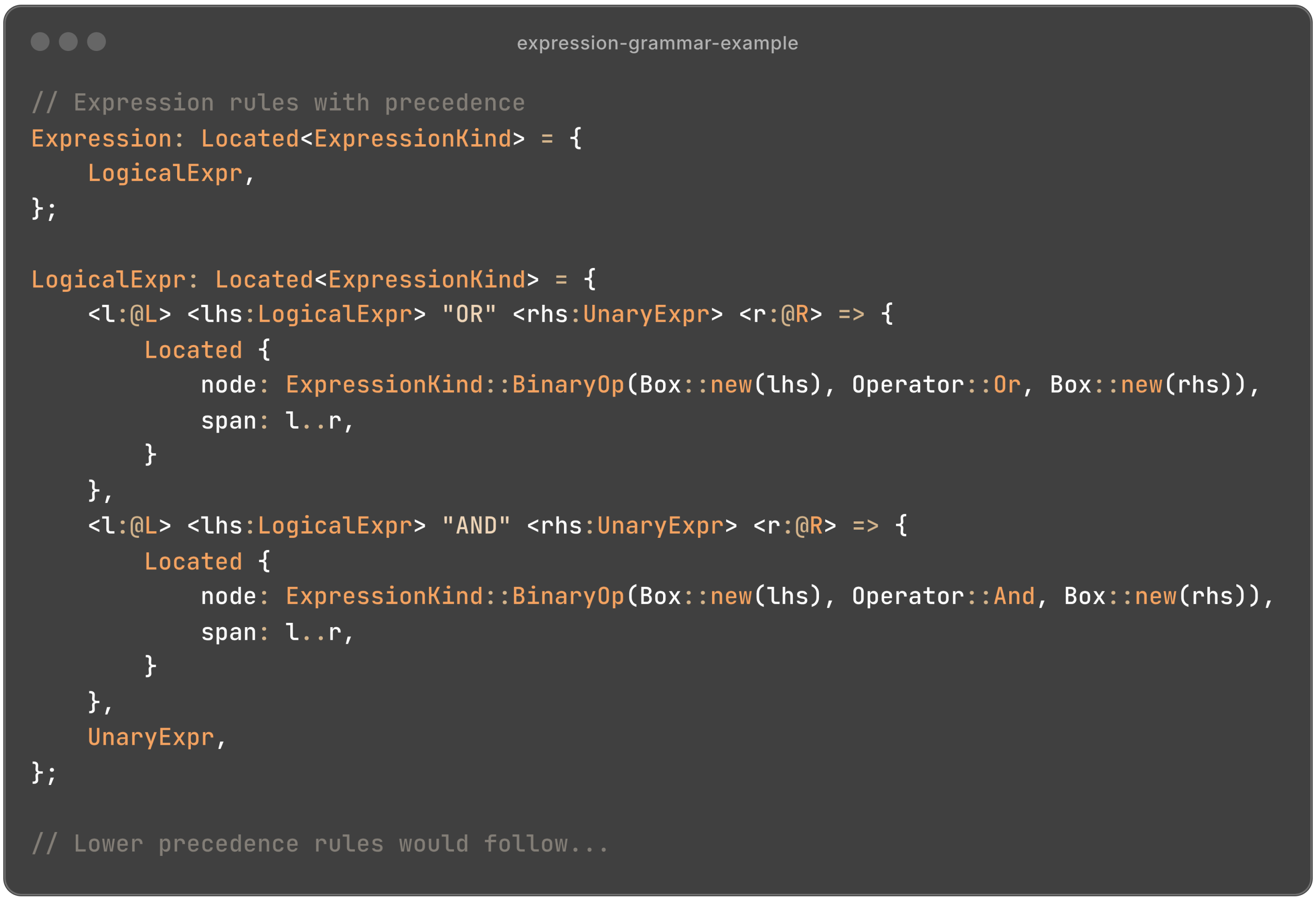
```rust
#[derive(Debug, Clone, PartialEq)]
pub struct Located<T> {
    pub node: T,
    pub span: Range<usize>,
}

#[derive(Debug, Clone, PartialEq)]
pub struct Program {
    pub name: String,
    pub declarations: Vec<Declaration>,
    pub statements: Vec<Statement>,
}

#[derive(Debug, Clone, PartialEq)]
pub enum StatementKind {
    Assignment(Expression, Expression),
    IfThen(Expression, Vec<Statement>),
    IfThenElse(Expression, Vec<Statement>, Vec<Statement>),
    DoWhile(Vec<Statement>, Expression),
    For(Expression, Expression, Expression, Expression, Vec<Statement>),
    Input(Expression),
    Output(Vec<Expression>),
}

#[derive(Debug, Clone, PartialEq)]
pub enum ExpressionKind {
    Identifier(String),
    ArrayAccess(String, Box<Expression>),
    Literal(Literal),
    BinaryOp(Box<Expression>, Operator, Box<Expression>),
    UnaryOp(UnaryOperator, Box<Expression>),
}
```

Figure 10: Core AST Structures

Each element in the AST is wrapped in a `Located<T>` structure that contains both the node and its position in the source code.

4.5 Expression Grammar

MiniSoft's expression grammar defines how expressions are parsed with proper operator precedence:



expression-grammar-example

```
// Expression rules with precedence
Expression: Located<ExpressionKind> = {
    LogicalExpr,
};

LogicalExpr: Located<ExpressionKind> = {
    <l:@L> <lhs:LogicalExpr> "OR" <rhs:UnaryExpr> <r:@R> => {
        Located {
            node: ExpressionKind::BinaryOp(Box::new(lhs), Operator::Or, Box::new(rhs)),
            span: l..r,
        }
    },
    <l:@L> <lhs:LogicalExpr> "AND" <rhs:UnaryExpr> <r:@R> => {
        Located {
            node: ExpressionKind::BinaryOp(Box::new(lhs), Operator::And, Box::new(rhs)),
            span: l..r,
        }
    },
    UnaryExpr,
};

// Lower precedence rules would follow...
```

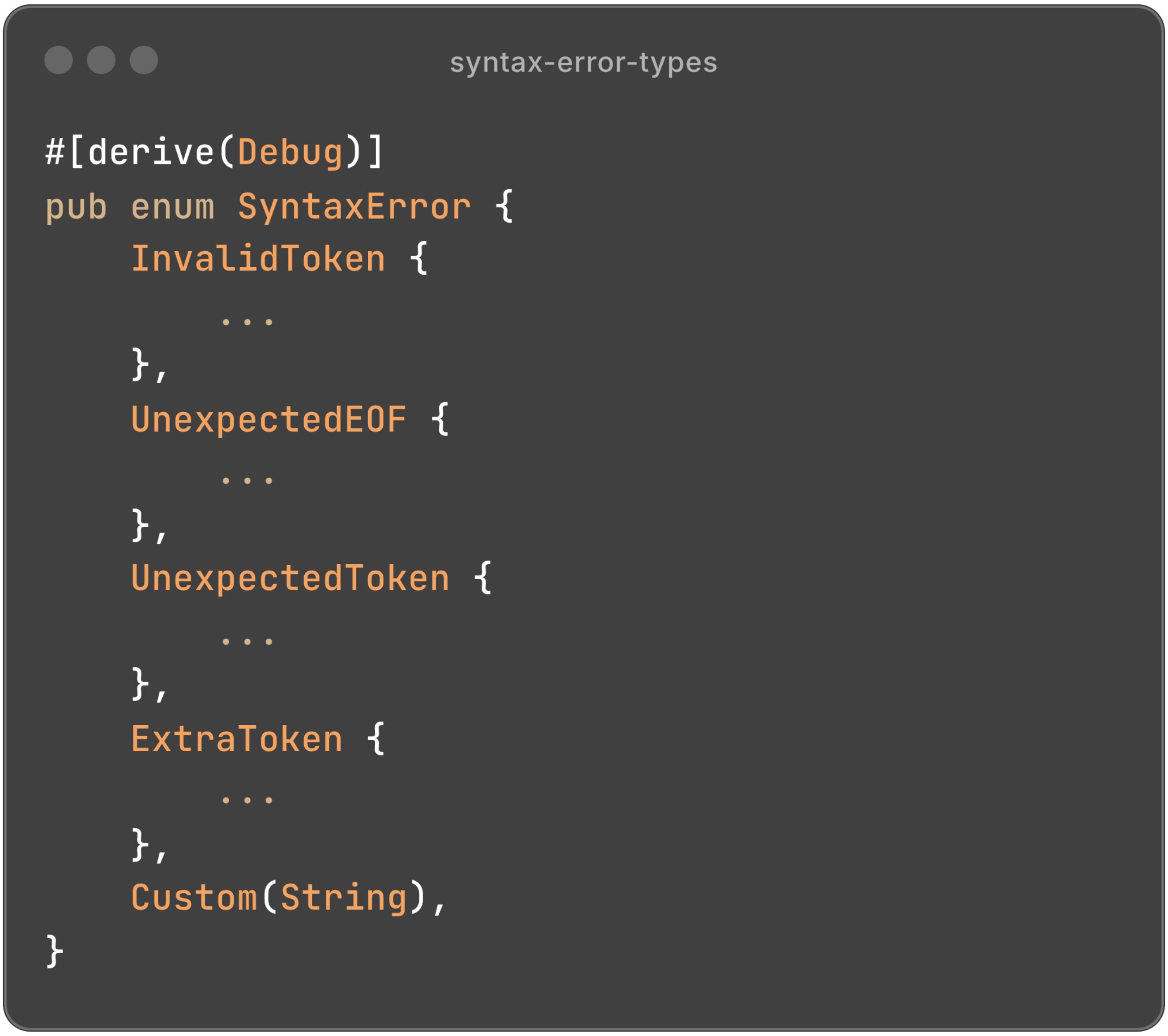
Figure 11: Expression Grammar Structure

This grammar ensures operators are evaluated in the correct precedence order, from highest to lowest:

1. Parenthesized expressions and primary expressions
2. Multiplicative operators (*, /)
3. Additive operators (+, -)
4. Comparison operators (==, !=, <, >, <=, >=)
5. Unary operators (!)
6. Logical operators (AND, OR)

4.6 Error Handling and Reporting

The parser detects and reports the first encountered syntax error with detailed feedback, then immediately terminates compilation to ensure prompt error resolution.



syntax-error-types

```rust
#[derive(Debug)]
pub enum SyntaxError {
    InvalidToken {
        ...
    },
    UnexpectedEOF {
        ...
    },
    UnexpectedToken {
        ...
    },
    ExtraToken {
        ...
    },
    Custom(String),
}
```

Figure 12: Syntax Error Types

A detailed example of the output is provided in the error handling section

5 Semantic Analysis

5.1 Overview of Semantic Analysis

Semantic analysis ensures that the program follows logical rules beyond syntax. In MiniSoft, this phase performs:

- Type checking for operations and assignments
- Identification of undeclared or multiply-declared variables
- Validation of constant integrity
- Detection of array bounds violations
- Recognition of potential runtime errors at compile time
- Verification of control flow constructs

5.2 Symbol Table Management

The symbol table tracks program identifiers and their attributes:

Listing 2: Symbol Table Structure

```rust
#[derive(Debug, Clone, PartialEq)]
pub enum SymbolKind {
    Variable,
    Constant,
    Array(usize),
}

#[derive(Debug, Clone)]
pub struct Symbol {
    pub name: String,           // Identifier name
    pub kind: SymbolKind,       // Variable, constant, or array
    pub symbol_type: Type,      // Data type (Int, Float, etc.)
    pub value: SymbolValue,     // Current value (if known at compile time)
    pub is_constant: bool,      // Whether the symbol can be modified
    pub line: usize,            // Declaration line number
    pub column: usize,          // Declaration column number
}
```

5.3 Semantic Analyzer Modules

The semantic analyzer delegates specific tasks to three specialized modules:

5.3.1 Declaration Analyzer

Responsibilities:

- Handles variable, array, and constant declarations
- Validates and registers entities in the symbol table

Key Functions:

- `handle_variable_declaration`: Validates variables without initial values
- `handle_variable_declaration_with_init`: Validates initialized variables
- `handle_constant_declaration`: Validates constants
- `handle_array_declaration`: Validates array declarations
- `handle_array_declaration_with_init`: Validates initialized arrays

Errors Handled:

- Duplicate declarations
- Type mismatches
- Array size mismatches

5.3.2 Statement Analyzer

Responsibilities:

- Analyzes program statements
- Handles assignments, control flow, and I/O operations

Key Functions:

- `handle_assignment`: Validates assignments
- `handle_condition`: Analyzes control flow conditions
- `handle_scope`: Analyzes statement blocks
- `handle_forloop`: Analyzes for loops
- `handle_input/output`: Validates I/O operations

Errors Handled:

- Type mismatches
- Invalid identifiers
- Non-boolean conditions
- Constant modification

5.3.3 Expression Analyzer

Responsibilities:

- Analyzes expressions
- Handles literals, identifiers, and operations

Key Functions:

- `handle_identifier`: Validates identifiers
- `handle_array_access`: Validates array accesses
- `handle_literal`: Processes literals
- `handle_binary/unary_operation`: Analyzes operations

Errors Handled:

- Undeclared identifiers
- Array bounds errors
- Type mismatches
- Division by zero

5.3.4 Integration Workflow

1. First pass: Analyze declarations (populate symbol table)
2. Second pass: Analyze statements (semantic validation)

The modular design ensures organized, maintainable analysis with clear separation of concerns.

5.4 Type System Implementation

MiniSoft features a static type system enforcing type compatibility at compile time:

Listing 3: Type System

```
#[derive(Debug, Clone, PartialEq)]
pub enum Type {
    Int,
    Float,
    String,
}

impl Type {
    pub fn is_compatible_with(&self, target: &Type) -> bool {
        match (self, target) {
            // Same types are always compatible
            (Type::Int, Type::Int) => true,
            (Type::Float, Type::Float) => true,
            (Type::String, Type::String) => true,

            // All other combinations are incompatible
            _ => false,
        }
    }
}
```

5.5 Expression Evaluation

The `expression_analyzer` module evaluates and analyzes expressions during semantic analysis, ensuring semantic validity, type rule adherence, and performing constant folding where possible.

Key Concepts

- **Expression Types**:
  - Literals (constants like integers, floats, strings)
  - Identifiers (variables/constants by name)
  - Array Access (elements accessed by index)
  - Binary Operations (two operands, e.g., +, -)
  - Unary Operations (single operand, e.g., !)
- **ValueType**:
  - `typ`: Expression type (Int, Float)
  - `value`: Evaluated value (if compile-time known)

Evaluation Workflow

- `analyze_expression`: Dispatches to appropriate handler:
  - Identifiers → `handle_identifier`
  - Array Access → `handle_array_access`
  - Literals → `handle_literal`
  - Binary Ops → `handle_binary_operation`
  - Unary Ops → `handle_unary_operation`

Expression Handlers

1. **Identifiers**
   - Function: `handle_identifier`
   - Purpose: Validates existence in symbol table
   - Errors: Undeclared identifier
   - Example:

```
let x = 5;  // x resolves to Int with value 5
```

2. **Array Access**
   - Function: `handle_array_access`
   - Errors: Undeclared, OOB, non-array, index type
   - Example:

```
let arr = [1, 2, 3];
let x = arr[1];  // Int with value 2
```

3. **Literals**
   - Function: `handle_literal`
   - Example:

```
let x = 42;    // Int(42)
let y = 3.14;  // Float(3.14)
```

4. **Binary Operations**
   - Function: `handle_binary_operation`
   - Operators: +, -, *, /, <, >, ==, !=, &&, ||
   - Errors: Type mismatch, division by zero
   - Example:

```
let x = 5 + 3;   // Int(8)
let y = 10 / 0;  // Error
```

5. **Unary Operations**
   - Function: `handle_unary_operation`
   - Operators: !
   - Errors: Invalid logical values
   - Example:

```
let x = !1;  // Int(0)
let y = !5;  // Error
```

Constant Folding

Evaluates constant expressions at compile time:

- `5 + 3` → `8`
- `10 / 2` → `5`

5.6 Error Detection and Reporting

The semantic analyzer performs comprehensive validation, detecting and collecting all semantic errors (type mismatches, undefined variables, scope violations, etc.) before reporting them collectively to allow for batch fixes.

Listing 4: Semantic Error Types

```rust
#[derive(Debug)]
pub enum SemanticError {
    ArraySizeMismatch { name: String, expected: usize, actual: usize },
    UndeclaredIdentifier { name: String },
    DuplicateDeclaration { name: String, original_line: usize },
    TypeMismatch { expected: String, found: String, context: Option<String>
       },
    DivisionByZero { },
    ConstantModification { name: String },
    ArrayIndexOutOfBounds { name: String, index: usize, size: usize },
    // Other error types...
}
```

Example of error output:

Listing 5: Semantic Error Output

```
Semantic Error: Type mismatch in assignment: expected Int, found Float
--> line 12, column 3
  |
12 |   result := average;
  |   ^^^^^^
Suggestion: Make sure the types match. Try converting from 'Float' to 'Int'
```

6 Code Generation

6.1 Intermediate Representation

The MiniSoft compiler uses quadruples as an intermediate representation that bridges the gap between the AST and machine-level instructions:

Listing 6: Quadruple Structure

```rust
#[derive(Debug, Clone, PartialEq)]
pub enum Operation {
    // Arithmetic operations
    Add, Subtract, Multiply, Divide,

    // Assignment and memory operations
    Assign, ArrayStore, ArrayLoad,

    // Control flow operations
    Label(usize), Jump(usize), JumpIfTrue(usize), JumpIfFalse(usize),

    // Comparison and logical operations
    Equal, NotEqual, LessThan, And, Or, Not,

    // I/O operations
    Input, Output,
}

```

```
#[derive(Debug, Clone, PartialEq)]
pub enum Operand {
    IntLiteral(i32),
    FloatLiteral(f32),
    StringLiteral(String),
    Variable(String),
    TempVariable(String),
    ArrayElement(String, Box<Operand>),
    Empty,
}

#[derive(Debug, Clone, PartialEq)]
pub struct Quadruple {
    pub operation: Operation,
    pub operand1: Operand,
    pub operand2: Operand,
    pub result: Operand,
}
```

6.2 Code Generation Strategy

The MiniSoft compiler generates code through recursive traversal of the AST:

Listing 7: Expression Code Generation

```
fn generate_expression(&mut self, expr: &Expression) -> Operand {
    match &expr.node {
        // For simple identifiers, just return a reference
        ExpressionKind::Identifier(name) => Operand::Variable(name.clone()),

        // For binary operations, process both operands
        ExpressionKind::BinaryOp(left, op, right) => {
            let left_result = self.generate_expression(left);
            let right_result = self.generate_expression(right);
            let result = self.program.new_temp();

            // Map AST operator to quadruple operation
            let operation = match op {
                Operator::Add => Operation::Add,
                Operator::Subtract => Operation::Subtract,
                // Other operators...
            };

            self.program.add(Quadruple {
                operation,
                operand1: left_result,
                operand2: right_result,
                result: result.clone(),
            });

            result
        },

        // Other expression types...
    }
}
```

For control structures like if-statements, the code generator creates labels and jumps:

Listing 8: If-Statement Code Generation

```
// Example of if-then-else statement
match &statement.node {
    StatementKind::IfThenElse(condition, then_block, else_block) => {
        let else_label = self.program.new_label();
        let end_label = self.program.new_label();

        // Evaluate condition
        let cond_result = self.generate_expression(condition);

        // Jump to else if condition is false
        self.program.add(Quadruple {
            operation: Operation::JumpIfFalse(else_label),
            operand1: cond_result,
            operand2: Operand::Empty,
            result: Operand::Empty,
        });

        // Generate code for then block
        // ...
    }
}
```

6.3 Example of Generated Code

For a simple MiniSoft program with a loop:

Listing 9: Sample MiniSoft Program

```
MainPrgm LoopExample;
Var
  let i: Int;
  let sum: Int;
BeginPg
{
  i := 1;
  sum := 0;

  while (i <= 10) do {
    sum := sum + i;
    i := i + 1;
  }

  output(sum);  // Outputs: 55
}
EndPg;
```

The generated quadruples would be:

Listing 10: Generated Quadruples

```
(ASSIGN, 1, _, i)              // i := 1
(ASSIGN, 0, _, sum)           // sum := 0
(LABEL_1, _, _, _)            // Start of loop
(LE, i, 10, t1)               // Compare i <= 10
(JMPF_2, t1, _, _)            // Jump to label 2 if false
```

```
(ADD, sum, i, t2)           // Calculate sum + i
(ASSIGN, t2, _, sum)        // sum := sum + i
(ADD, i, 1, t3)             // Calculate i + 1
(ASSIGN, t3, _, i)          // i := i + 1
(JUMP_1, _, _, _)           // Jump back to start
(LABEL_2, _, _, _)          // End of loop
(OUTPUT, sum, _, _)         // Output sum
```

7 Error Handling

7.1 Error Categories

The MiniSoft compiler implements comprehensive error detection across all compilation phases:

1. **Lexical Errors**: Invalid characters or token formation issues
2. **Syntax Errors**: Violations of grammar rules
3. **Semantic Errors**: Logically invalid constructs
4. **Code Generation Errors**: Issues in the final translation phase

7.2 Error Reporting Framework

The compiler uses a unified error reporting framework based on the `ErrorReporter` trait:

Listing 11: Error Reporter Interface

```
pub trait ErrorReporter {
    fn report(&self, source_code: Option<&str>) -> String;
    fn get_suggestion(&self) -> Option<String>;
    fn get_error_name(&self) -> String;
    fn get_location_info(&self) -> (usize, usize);
}
```

This ensures consistent error presentation across all compiler phases.

7.3 Source Code Context

Error messages include source code context with visual indicators:

Listing 12: Error with Source Context

```
Syntax Error: Unexpected token '}'
--> line 15, column 3
  |
15 | if (x > 10) then {
  |   output("Value too large");
  | }
  |   ^
Expected one of: ';'
Suggestion: Missing semicolon at the end of statement before this closing
   brace
```

7.4 Example Error Scenarios

The compiler detects various error types:

Listing 13: Lexical Error Example

```
Lexical Error: Identifier 'veryLongIdentifier' exceeds maximum length of 14
   characters
--> line 3, column 7
  |
3 |   let veryLongIdentifier: Int;
  |        ^^^^^^^^^^^^^^^^^^
Suggestion: Identifiers must be 14 characters or less
```

Listing 14: Semantic Error Example

```
Semantic Error: Type mismatch in assignment: expected Int, found String
--> line 7, column 3
  |
7 |   x := message;
  |   ^
Suggestion: Cannot assign a String value to an Int variable.
```

Listing 15: Compile-Time Error Example

```
Semantic Error: Division by zero detected in constant expression
--> line 7, column 14
  |
7 |   result := 10 / DIVISOR;
  |                ^^^^^^^^^
Suggestion: Check the divisor value to ensure it is not zero
```

8 Testing and Validation

8.1 Test Methodology

The MiniSoft compiler was tested using a comprehensive suite of test cases designed to verify functionality across all compilation phases. Tests were developed using both black-box and white-box approaches to ensure complete coverage.

8.2 Test Cases

The test suite includes unit tests for individual components and integration tests for the complete compilation pipeline:

9 Results and Evaluation

9.1 Functionality Assessment

The MiniSoft compiler successfully implements all required language features and compilation phases. It correctly handles a wide range of program constructs while providing meaningful error messages for invalid code.

9.2 Example Compilation

Here is a complete compilation example of a factorial program:

Listing 16: Sample MiniSoft Program

```
MainPrgm Factorial;
Var
  let n: Int;
  let result: Int;
BeginPg
{
  n := 5;
  result := 1;

  for i from 1 to n step 1 {
    result := result * i;
  }

  output(result);  // Outputs: 120
}
EndPg;
```

10 Conclusion

10.1 Achievements

The MiniSoft compiler successfully implements a complete compilation pipeline that translates MiniSoft source code into quadruples format. Key achievements include:

- A robust lexical analyzer using Logos
- A comprehensive syntax analyzer using LALRPOP
- A thorough semantic analyzer with type checking
- An intermediate code generator using quadruples
- A detailed error reporting system

10.2 Challenges

Challenges encountered during implementation included:

- Designing an error recovery strategy that balances robustness with usability
- Implementing proper type checking for a statically typed language
- Handling complex control flow structures in code generation
- Creating clear, actionable error messages

10.3 Future Improvements

Potential future enhancements include:

- Add executable generation (binary/bytecode output)
- Additional optimization passes
- Support for functions and procedures
- Enhanced type system with type conversion
- Improved warning system for potential issues
- IDE integration for interactive development

References

[1] The Rust Programming Language: `https://www.rust-lang.org/`

[2] Logos: `https://logos.maciej.codes/`

[3] LALRPOP: `https://lalrpop.github.io/lalrpop/`

[4] Cranelift: `https://docs.rs/cranelift/latest/cranelift/`